UNITED STATES PATENT APPLICATION
FOR

PROGRAM OBJECT READ BARRIER

INVENTORS:

ALI-REZA ADL-TABATABAI,
a citizen of the United States of America

JAYASHANKAR BHARADWAJ,
a citizen of the United States of America

TATIANA SHPEISMAN,
a citizen of Russia

PREPARED BY:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP
12400 WILSHIRE BOULEVARD
SEVENTH FLOOR
LOS ANGELES, CA 90025-1030
(303) 740-1980

EXPRESS MAIL CERTIFICATE OF MAILING
"Express Mail" Mailing No. EL 981993198 US

I hereby certify that I am causing the above-referenced correspondence to be deposited with the United States Postal Service "Express Mail Post Office to Addressee" service on the date indicated below and that this paper or fee has been addressed to the Commissioner for Patents, Patent Application, P.O. Box 1450, Alexandria, VA 22313.
Date of Deposit: March 31, 2004
Name of Person Mailing Correspondence: Krista Mathieson

Krista Mathieson — Signature
March 31, 2004 — Date

PROGRAM OBJECT READ BARRIER

FIELD

[0001] An embodiment of the invention relates to computer operations in general, and more specifically to a program object read barrier.

BACKGROUND

[0002] In computer software, a read barrier mechanism may exist for a program object. In this type of operation, the program object is said to be guarded, and any read access to the program object is then trapped or detected.

[0003] A read barrier may have many applications in software, including: (1) A read barrier may be used to mediate access to proxies that represent remote objects in distributed object systems. Similarly, read barriers may be used to mediate accesses to persistent objects in persistent object systems. (2) A read barrier may be used to trap accesses to objects that are being garbage collected in runtimes that employ concurrent garbage collection. (3) A read barrier may be used to track references across memory heap regions in runtimes that partition the heap into regions. This may include generational garbage collectors, which partition a heap into generations. (4) A read barrier may be used to implement debug watch points.

[0004] If a read barrier has been implemented for a program object, a runtime environment checks each read access to determine whether the access touches a guarded object. However, the process of providing a read barrier may have negative effects on performance. The requirement to provide a check of each read access to determine

whether the access is directed to a guarded object may consume considerable computer resources, and thus may impose costs for system operations.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] The invention may be best understood by referring to the following description and accompanying drawings that are used to illustrate embodiments of the invention. In the drawings:

[0006] **Figure 1** illustrates an embodiment of a read barrier mechanism;

[0007] **Figure 2** is a flowchart illustrating an embodiment of a read barrier operation;

[0008] **Figure 3** illustrates an embodiment of a read barrier check using control speculation;

[0009] **Figure 4** illustrates an embodiment of a combination of a null pointer check and a read barrier check using control speculation;

[0010] **Figure 5** illustrates an embodiment of a read barrier for a single-byte access; and

[0011] **Figure 6** illustrates an embodiment of a computer environment.

DETAILED DESCRIPTION

[0012] A method and apparatus are described for a program object read barrier.

[0013] According to an embodiment of the invention, a mechanism and process for implementing a read barrier is provided. A read barrier traps access to a program object. An object may be referred to as being guarded if access to it results in trapping. To implement read barriers, a computer system runtime guards all read accesses to an object with a check to determine whether an access touches a guarded object.

[0014] Under an embodiment of the invention, a control speculation feature is utilized to implement an object read barrier. An embodiment of the invention may be implemented in managed runtime environments (MRTEs), with such environments including the Java Virtual Machine or the .NET Common Language Runtime. The embodiment of the invention implements a read barrier without increasing overhead by combining the read barrier with a null reference check, which is generally required for a MRTE. An embodiment of the invention may be implemented in a compiler, including the Intel StarJIT dynamic compiler, a just-in-time compiler for both Java and .NET. A just-in-time (JIT) compiler is a program that dynamically converts program code into machine executable code or instructions at the request of the execution environment.

[0015] Under an embodiment of the invention, a processor architecture may supports control speculation by providing two instructions, the instructions being a speculative load instruction (which may, for example, be designated as ld.s) and a speculation check instruction (which may, for example, be designated as chk.s). A processor utilized in an embodiment of the invention may include an Intel Itanium or Intel Itanium 2 processor. Under an embodiment of the invention, a processor does not

generate a fault if a speculative load (ld.s) causes a hardware exception, such as a misaligned access exception. Instead, the processor invalidates the result of the load by making a particular setting, such as setting the NaT (not a thing) bit representing the 65th bit of an integer register for the Intel Itanium Processor Family (IPF) architecture. The speculation-check instruction (chk.s) checks the NaT bit and branches to a recovery code if the speculative load fails, indicated by the NaT bit being set. The control speculation feature generally allows a compiler to schedule a load speculatively above program branches upon which the load is control dependent. Under an embodiment of the invention, a control speculation feature or process is further used to implement read barriers in a managed runtime environment.

[0016] Under an embodiment of the invention, a read barrier check is implemented by using pointer swizzling and by combining the read barrier check with a null reference check, which is generally required in an MRTE. Pointer swizzling refers to converting external designations (names, array indices, or references) within a data structure into address pointers when the data structure is brought into memory. Under an embodiment of the invention, in order to guard an object the runtime (such as in the form of a system garbage collector) rewrites a pointer to the guarded object in some manner, such as setting the least-significant bit of its address. In this manner, a non-byte-sized access via the pointer will cause a misaligned access exception. An embodiment of the invention uses control speculative loads to defer and process misaligned access exceptions that then will result from the guarded object reads. Further, to avoid misaligned exceptions that are the result of guarded object writes, a compiler may

precede each object write with an instruction that explicitly clears the least-significant bit of the object address.

[0017] An MRTE generally requires a load of an object data to raise a null reference exception if its base object reference is null. Under an embodiment of the invention, a compiler combines a null pointer check and a read barrier check into a single speculative load. The read barrier combined with the null pointer check thus avoids imposing additional overhead on program execution because run-time null pointer checks generally are already required for object accesses, unless a compiler can statically prove that the object reference is non-null.

[0018] A byte-size access would not cause a misaligned access exception without modification. Under an embodiment of the invention, read barriers may be implemented for byte-size object fields in a modified process. According to a first embodiment, a compiler may generate code that explicitly checks the least-significant bit of each object address to determine if the object is guarded, and, if so, the code executes the read barrier and adjusts the field address. This technique can also be used to implement read barrier checks for arrays of bytes. According to a second embodiment, a compiler may implements a byte-size object field access as a two-byte load, which then reads the required byte and the next byte if the byte field offset is even or the required byte and the previous byte if the byte field offset is odd. The compiler also generates an additional shift instruction to place the loaded byte into the least-significant byte of the register if the object offset is even and the memory layout is big endian (the most significant value in the sequence is stored at the lowest storage address), or if the object offset is odd and the memory layout is little endian (least significant value in the

sequence is stored first). Under an embodiment of the invention, the extra shift instruction can be avoided if runtime pads the object so that byte-size fields are always aligned on the proper boundary.

[0019] **Figure 1** provides a figurative illustration of an embodiment of a read barrier mechanism. In this illustration, a first access request **105**, such as from a first program object **110**, is made for a second program object **115**. In this general illustration, if the second program object **115** is guarded, the access request **105** is trapped, indicated by the read barrier result **120**. Under an embodiment of the invention, a second read request **125** is made by a third program object **130** for a fourth program object **135**. In this embodiment, a combined null reference and read barrier check **140** is utilized, which may provide the read barrier mechanism without imposing additional overhead when the null reference check is already required. There is then a determination whether a null reference condition **145** or a read barrier condition **150** exists.

[0020] **Figure 2** is a flowchart illustrating an embodiment of a read barrier operation. In this example, a read request is received **205**. In response to the read request, a speculative load is performed **210**, followed by a speculation check **215**. In this illustration, the use of the speculative load and the speculation check implements a read barrier check. If the speculative load succeeds, the process continues with the next instruction **220**. If the speculative load fails, there is a branch to recovery code **225**. In the recovery code, there is a determination whether a read barrier is needed **230**. This determination may be made by determining whether, for example, the least significant bit of the address is set. If a read barrier is needed, indicating that a program object is guarded, a read barrier process is engaged **235**. If a read barrier is not needed, a

non-speculative load is performed **240** and there is a branch back to the next instruction **220**.

[0021] **Figure 3** illustrates an embodiment of code **305** for non-byte-size access using control speculation. In Figure 3, sample pseudo-code is provided to illustrate an example of reading an object field p.x, where p is an object reference and x is a field within this object. In this example, instruction I1 **310** computes the address of the field x by adding the offset of x to the object's base reference (variable p). The read barrier check and field access are implemented as a speculative load, instruction I2 **315**, shown as the operation ld.s. This is followed by a speculation check, instruction I3 **320**. When an object reference for a guarded object has been swizzled, the speculative load will fail (for example, the target register's NaT bit may be set) and the instruction then transfers control to a recovery code **330**. The recovery code **330** first checks if the object access requires a read barrier by testing the least-significant bit of the field address by instruction I4 **335**. If the least-significant bit is set, the last bit of the address will be cleared, as shown by instruction I5 **340**, and the read barrier in executed, instruction I6 **345**. Instruction I7 **350** provides for re-executing the load non-speculatively. Instruction I8 **355** then branches back to the main instruction sequence at NextInst **325**. If the only reason for speculative load failure is access to a guarded object, i.e., if a processor is configured such that a speculative load does not fail on any serviceable fault (such as a dual transition lookaside buffer, or DTLB, miss) other than a misaligned access, then the recovery code can simply handle the read barrier without testing the least-significant bit of the address.

[0022] **Figure 4** illustrates an embodiment of a combination of a null pointer check and a read barrier check **405**. In Figure 4, sample pseudo-code is provided to illustrate an example of reading an object field p.x, where p is an object reference and x is a field within this object. In this example, instruction I1 **410** computes the address of the field x by adding the offset of x to the object's base reference (variable p). A null reference check, read barrier check, and field access are implemented as a single speculative load, instruction I2 **415**, followed by speculation check, instruction I3 **420**. When an object reference is null or the guarded object reference has been swizzled, the speculative load fails and the check instruction transfers control to recovery code **430**.

[0023] The recovery code compares the address of the load with the field offset, instruction I4 **435**, to check if the failure was due to a null reference. If the object base is null, the recovery code transfers the control to the code that raises null pointer exceptions, **440**. If the object base is not null, the recovery code **430** proceeds to check if the object access requires a read barrier by testing the least-significant bit of the field address, instruction I6 **445**. If the least-significant bit is set, the last bit of the address will be cleared, instruction I7 **450**, and the read barrier in executed, instruction I8 **455**. Instruction I9 **460** re-executes the load non-speculatively. Instruction I10 **465** branches back to the main instruction sequence at NextInst **425**. If a processor is configured such that a speculative load does not fail on a serviceable fault other than misalignment, the recovery code can perform a single test, which may be either the null pointer reference test or the guarded object access test, to determine the reason for the speculative load failure.

[0024] **Figure 5** illustrates an embodiment of a single byte access. In figure 5, a read barrier check for a single-byte field with an odd offset **505** is illustrated, assuming little-endian memory layout. Similar processes for even offset and big-endian memory layout combinations may also be illustrated. Instruction I1 **510** computes the address of the byte preceding field x in the object layout by adding the offset of the field x minus one to the object's base reference. A speculative load, instruction I2 **515**, reads two bytes from this address. Because the offset of the field x is odd the value added to the object's base reference is even, so the load I2 causes misaligned access exception if and only if the object reference has been swizzled. Instruction I3 **520** shifts the loaded value of the field x into the least significant byte of the register. Instruction I4 **525** performs a speculation check. If the speculative load fails, the control is transferred to recovery code **535**. The recovery code first checks if the object access requires a read barrier by testing the least-significant bit of the field address, instruction I5 **540**. If the least-significant bit is set, the code executes the read barrier, instruction I6 **545**. In this case variable t contains the address of field x rather than the address of the previous byte. Single-byte load instruction I7 **550** non-speculatively reads the value of the field x and instruction I8 **555** branches back to the main instruction sequence at label NextInst **530**.

[0025] Techniques described here may be used in many different environments. **Figure 6** is block diagram of an exemplary computer system environment. Under an embodiment of the invention, a computer **600** comprises a bus **605** or other communication means for communicating information, and a processing means such as one or more processors **610** (shown as processor 1 **611** , processor 2 **612**,

and continuing through processor *n* **613**) coupled with the first bus **605** for processing information.

[0026] The computer **600** further comprises a random access memory (RAM) or other dynamic storage device as a main memory **615** for storing information and instructions to be executed by the processors **610**. The instructions may include instructions related to a program compiler. Main memory **615** also may be used for storing temporary variables or other intermediate information during execution of instructions by the processors **610**. The computer **600** also may comprise a read only memory (ROM) **620** and/or other static storage device for storing static information and instructions for the processor **610**.

[0027] A data storage device **625** may also be coupled to the bus **605** of the computer **600** for storing information and instructions. The data storage device **625** may include a magnetic disk or optical disc and its corresponding drive, flash memory or other nonvolatile memory, or other memory device. Such elements may be combined together or may be separate components, and utilize parts of other elements of the computer **600**. Under an embodiment of the invention, the data storage device **625** may include storage of a program compiler for use on the computer **600**. The data storage device **625** may include storage of program code that includes one or more program object read barriers in operation.

[0028] The computer **600** may also be coupled via the bus **605** to a display device **630**, such as a liquid crystal display (LCD) or other display technology, for displaying information to an end user. In some environments, the display device may be a touch-screen that is also utilized as at least a part of an input device. In some

environments, display device **630** may be or may include an auditory device, such as a speaker for providing auditory information. An input device **640** may be coupled to the bus **605** for communicating information and/or command selections to the processor **610**. In various implementations, input device **640** may be a keyboard, a keypad, a touch-screen and stylus, a voice-activated system, or other input device, or combinations of such devices. Another type of user input device that may be included is a cursor control device **645**, such as a mouse, a trackball, or cursor direction keys for communicating direction information and command selections to processor **610** and for controlling cursor movement on display device **630**.

[0029] A communication device **650** may also be coupled to the bus **605**. Depending upon the particular implementation, the communication device **650** may include a transceiver, a wireless modem, a network interface card, or other interface device. The computer **600** may be linked to a network or to other devices using the communication device **650**, which may include links to the Internet, a local area network, or another environment. In an embodiment of the invention, the communication device **650** may provide a link to a service provider over a network.

[0030] In the description above, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention may be practiced without some of these specific details. In other instances, well-known structures and devices are shown in block diagram form.

[0031] The present invention may include various processes. The processes of the present invention may be performed by hardware components or may be embodied in

machine-executable instructions, which may be used to cause a general-purpose or special-purpose processor or logic circuits programmed with the instructions to perform the processes. Alternatively, the processes may be performed by a combination of hardware and software.

[0032] Portions of the present invention may be provided as a computer program product, which may include a machine-readable medium having stored thereon instructions, which may be used to program a computer (or other electronic devices) to perform a process according to the present invention. The machine-readable medium may include, but is not limited to, floppy diskettes, optical disks, CD-ROMs (compact disk read-only memory), and magneto-optical disks, ROMs (read-only memory), RAMs (random access memory), EPROMs (erasable programmable read-only memory), EEPROMs (electrically-erasable programmable read-only memory), magnet or optical cards, flash memory, or other type of media / machine-readable medium suitable for storing electronic instructions. Moreover, the present invention may also be downloaded as a computer program product, wherein the program may be transferred from a remote computer to a requesting computer by way of data signals embodied in a carrier wave or other propagation medium via a communication link (e.g., a modem or network connection).

[0033] Many of the methods are described in their most basic form, but processes can be added to or deleted from any of the methods and information can be added or subtracted from any of the described messages without departing from the basic scope of the present invention. It will be apparent to those skilled in the art that many further modifications and adaptations can be made. The particular embodiments are not

provided to limit the invention but to illustrate it. The scope of the present invention is not to be determined by the specific examples provided above but only by the claims below.

[0034] It should also be appreciated that reference throughout this specification to "one embodiment" or "an embodiment" means that a particular feature may be included in the practice of the invention. Similarly, it should be appreciated that in the foregoing description of exemplary embodiments of the invention, various features of the invention are sometimes grouped together in a single embodiment, figure, or description thereof for the purpose of streamlining the disclosure and aiding in the understanding of one or more of the various inventive aspects. This method of disclosure, however, is not to be interpreted as reflecting an intention that the claimed invention requires more features than are expressly recited in each claim. Rather, as the following claims reflect, inventive aspects lie in less than all features of a single foregoing disclosed embodiment. Thus, the claims are hereby expressly incorporated into this description, with each claim standing on its own as a separate embodiment of this invention.